Java进阶-泛型

> 本预先资料来源于Oracle官方文档Java™ 教程-Java Tutorials
> 官方文档:https://docs.oracle.com/javase/tutorial/java/generics/index.html
> 中文翻译:https://pingfangx.github.io/java-tutorials/java/generics/types.html

泛型（Generic）是Java编程语言的强大功能。它们提高了代码的类型安全性，使你在编译时可以检测到更多错误。

在任何不平凡的软件项目中，错误都是生活中的事实。仔细的计划，编程和测试可以帮助降低其普遍性，但是无论如何，它们总会在某种程度上找到爬入你的代码的方法。随着新功能的引入以及代码库的大小和复杂性的增加，这一点变得尤为明显。

幸运的是，某些错误比其它错误更容易发现。例如，可以在早期发现编译时错误；你可以使用编译器的错误消息找出问题所在，然后就可以在那里进行修复。但是，运行时错误可能会带来更多问题。它们并不总是立即浮出水面，而当它们浮出水面时，可能是在程序中与实际问题原因相去甚远的某个时刻。

泛型通过在编译时检测到更多错误来增加代码的稳定性。完成本课程后，你可能需要继续阅读Gilad Bracha的Generics教程。

Why Use Generics?（为什么要使用泛型？）

简而言之，泛型在定义类，接口和方法时使类型（类和接口）成为参数。与方法声明中使用的更熟悉的形式参数非常相似，类型参数为你提供了一种使用不同输入重复使用相同代码的方法。区别在于形式参数的输入是值，而类型参数的输入是类型。

与非泛型代码相比，使用泛型的代码具有许多优点：

- 在编译时进行更强的类型检查。Java编译器将强类型检查应用于通用代码，如果代码违反类型安全，则会发出错误。修复编译时错误比修复运行时错误容易，后者可能很难找到。
- 消除类型转换。以下不带泛型的代码段需要强制转换：

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);
```

当使用泛型重写时，代码不需要强制转换：

```
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0);   // no cast
```

- 使程序员能够实现通用算法。通过使用泛型，程序员可以实现对不同类型的集合进行工作，可以自定义并且类型安全且易于阅读的泛型算法。

Generic Types（通用类型）

通用类型是通过类型进行参数化的通用类或接口。下面的Box类将被修改以演示该概念。

A Simple Box Class（一个简单的Box类）

首先检查对任何类型的对象进行操作的非通用Box类。它只需要提供两种方法：`set`（将对象添加到Box中）和`get`（将其检索到）：

```
public class Box {
    private Object object;

    public void set(Object object) { this.object = object; }
    public Object get() { return object; }
}
```

由于它的方法接受或返回一个`Object`，因此只要它不是原始类型之一，你就可以随意传递任何想要的东西。在编译时无法验证类的使用方式。代码的一部分可能会将`Integer`放在Box中，并期望从中取出`Integer`，而代码的另一部分可能会错误地传入`String`，从而导致运行时错误。

A Generic Version of the Box Class（Box类的通用版本）

通用类的定义格式如下：

```
class name<T1, T2, ..., Tn> { /* ... */ }
```

在类名之后，类型参数部分由尖括号（`<>`）分隔。它指定了类型参数（也称为类型变量）T1、T2、...和Tn。

要更新Box类以使用泛型，可以通过将代码“`public class Box`”更改为“`public class Box`”来创建泛型类型声明。

进行此更改后，Box类变为：

```
/**
 * Generic version of the Box class.
 * @param <T> the type of the value being boxed
 */
public class Box<T> {
    // T stands for "Type"
    private T t;

    public void set(T t) { this.t = t; }
    public T get() { return t; }
}
```

如你所见，所有出现的`Object`都将替换为T。类型变量可以是你指定的任何非基本类型：任何类类型，任何接口类型，任何数组类型，甚至是另一个类型变量。

可以将相同的技术应用于创建通用接口。

Type Parameter Naming Conventions（类型参数命名约定）

按照约定，类型参数名称是单个大写字母。这与你已经知道的变量命名约定形成鲜明对比，并且有充分的理由：没有该约定，将很难分辨类型变量与普通类或接口名称之间的区别。

最常用的类型参数名称是：

- E - Element (Java Collections Framework广泛使用)
- K - Key
- N - Number
- T - Type

- V - Value
- S,U,V etc. - 2nd, 3rd, 4th types

你将看到在Java SE API以及本课程其余部分中使用的这些名称。

Invoking and Instantiating a Generic Type（调用和实例化泛型类型）

要从代码中引用通用Box类，必须执行通用类型调用，该调用将T替换为某些具体值，例如`Integer`：

```
Box<Integer> integerBox;
```

你可以认为泛型类型调用类似于普通方法调用，但是你没有将参数传递给方法，而是将类型参数（在这种情况下为`Integer`）传递给Box类本身。

> 类型参数和类型参数术语：许多开发人员可以互换使用术语“type parameter”（又名形参）和“type argument”（又名实参），但是这些术语并不相同。编码时，提供类型实参以创建参数化类型。因此，`Foo`中的T是类型参数（形参），而`Foo f`中的`String`是类型参数（实参）。在使用这些术语时，本课将遵循此定义。

像任何其它变量声明一样，此代码实际上不会创建新的Box对象。它只是声明integerBox将保存对“Box of Integer”的引用，这就是读取`Box`的方式。

泛型类型的调用通常称为参数化类型。

要实例化此类，请像往常一样使用`new`关键字，但将``放在类名和括号之间：

```
Box<Integer> integerBox = new Box<Integer>();
```

The Diamond（菱形）

在Java SE 7和更高版本中，只要编译器可以从上下文确定或推断出类型参数，就可以用一组空的类型参数（`<>`）替换调用通用类的构造函数所需的类型参数。这对尖括号`<>`被非正式地称为菱形。例如，你可以使用以下语句创建`Box`的实例：

```
Box<Integer> integerBox = new Box<>();
```

有关菱形符号和类型推断的更多信息，请参见类型推断。

Multiple Type Parameters（多种类型的参数）

如前所述，泛型类可以具有多个类型参数。例如，通用的OrderedPair类实现了通用的Pair接口：

```
public interface Pair<K, V> {
    public K getKey();
    public V getValue();
}

public class OrderedPair<K, V> implements Pair<K, V> {

    private K key;
    private V value;

    public OrderedPair(K key, V value) {
    this.key = key;
    this.value = value;
    }
```

```
15
16      public K getKey()   { return key; }
17      public V getValue() { return value; }
18  }
```

以下语句创建OrderedPair类的两个实例：

```
1  Pair<String, Integer> p1 = new OrderedPair<String, Integer>("Even", 8);
2  Pair<String, String>  p2 = new OrderedPair<String, String>("hello",
   "world");
```

代码`new OrderedPair`将K实例化为`String`，将V实例化为`Integer`。因此，OrderedPair的构造函数的参数类型分别为`String`和`Integer`。由于自动装箱，将`String`和`int`传递给类是有效的。

如The Diamond（菱形）所述，由于Java编译器可以从声明`OrderedPair`推断出K和V类型，因此可以使用菱形表示法来缩短这些语句：

```
1  OrderedPair<String, Integer> p1 = new OrderedPair<>("Even", 8);
2  OrderedPair<String, String>  p2 = new OrderedPair<>("hello", "world");
```

要创建通用接口，请遵循与创建通用类相同的约定。

Parameterized Types（参数化类型）

你还可以用参数化类型（即`List`）替换类型参数（即K或V）。例如，使用`OrderedPair`示例：

```
1  OrderedPair<String, Box<Integer>> p = new OrderedPair<>("primes", new
   Box<Integer>(...));
```

Raw Types（原始类型）

原始类型是没有任何类型参数的泛型类或接口的名称。例如，给定通用Box类：

```
1  public class Box<T> {
2      public void set(T t) { /* ... */ }
3      // ...
4  }
```

要创建`Box`的参数化类型，请为形式类型参数T提供一个实际的类型参数：

```
1  Box<Integer> intBox = new Box<>();
```

如果省略实际的类型参数，则创建`Box`的原始类型：

```
1  Box rawBox = new Box();
```

因此，Box是通用类型`Box`的原始类型。但是，非泛型类或接口类型不是原始类型。

原始类型显示在旧版代码中，因为在JDK 5.0之前，许多API类（例如`Collections`类）不是通用的。使用原始类型时，你实际上会获得泛型行为（Box为你提供对象）。为了向后兼容，允许将参数化类型分配给其原始类型：

```
Box<String> stringBox = new Box<>();
Box rawBox = stringBox;               // OK
```

但是，如果将原始类型分配给参数化类型，则会收到警告：

```
Box rawBox = new Box();           // rawBox is a raw type of Box<T>
Box<Integer> intBox = rawBox;     // warning: unchecked conversion
```

如果你使用原始类型来调用在相应的泛型类型中定义的泛型方法，也会收到警告：

```
Box<String> stringBox = new Box<>();
Box rawBox = stringBox;
rawBox.set(8);  // warning: unchecked invocation to set(T)
```

该警告表明原始类型会绕过通用类型检查，从而将不安全代码的捕获推迟到运行时。因此，应避免使用原始类型。

Type Erasure部分提供了有关Java编译器如何使用原始类型的更多信息。

Unchecked Error Messages（未检查的错误消息）

如前所述，将旧代码与通用代码混合时，你可能会遇到类似于以下内容的警告消息：

```
Note: Example.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

当使用旧的API操作原始类型时可能会出现这种情况，如下例所示：

```
public class WarningDemo {
    public static void main(String[] args){
        Box<Integer> bi;
        bi = createBox();
    }

    static Box createBox(){
        return new Box();
    }
}
```

术语“未检查”表示编译器没有足够的类型信息来执行确保类型安全所需的所有类型检查。尽管编译器会给出提示，但默认情况下禁用“未检查”警告。要查看所有“未检查”的警告，请使用`-Xlint:unchecked`重新编译。

使用`-Xlint:unchecked`重新编译前面的示例将显示以下附加信息：

```
WarningDemo.java:4: warning: [unchecked] unchecked conversion
found   : Box
required: Box<java.lang.Integer>
        bi = createBox();
                     ^
1 warning
```

要完全禁用未检查的警告，请使用`-Xlint:unchecked`标志。`@SuppressWarnings("unchecked")`注解禁止未检查的警告。如果你不熟悉`@SuppressWarnings`语法，请参阅注解。

Generic Methods（通用方法）

通用方法是指引入自己的类型参数的方法。这类似于声明一个泛型方法，但类型参数的范围仅限于声明它的方法。允许使用静态和非静态的泛型方法，也允许使用泛型类构造函数。

通用方法的语法包括类型参数列表，在尖括号内，该列表出现在方法的返回类型之前。对于静态泛型方法，类型参数部分必须出现在方法的返回类型之前。

Util类包含一个通用方法`compare`，该方法比较两个Pair对象：

```
public class Util {
    public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2) {
        return p1.getKey().equals(p2.getKey()) &&
               p1.getValue().equals(p2.getValue());
    }
}

public class Pair<K, V> {

    private K key;
    private V value;

    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public void setKey(K key) { this.key = key; }
    public void setValue(V value) { this.value = value; }
    public K getKey()   { return key; }
    public V getValue() { return value; }
}
```

调用此方法的完整语法为：

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");
Pair<Integer, String> p2 = new Pair<>(2, "pear");
boolean same = Util.<Integer, String>compare(p1, p2);
```

该类型已明确提供，如上所示。通常，可以将其忽略，编译器将推断出所需的类型：

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");
Pair<Integer, String> p2 = new Pair<>(2, "pear");
boolean same = Util.compare(p1, p2);
复制代码
```

此功能称为类型推断，使你可以在不指定尖括号之间的类型的情况下，将通用方法作为普通方法调用。“类型推断”将进一步讨论该主题。

Bounded Type Parameters（限定类型参数）

有时你可能想限制可以在参数化类型中用作类型参数的类型。例如，对数字进行操作的方法可能只希望接受`Number`或其子类的实例。这就是限定类型参数的用途。

要声明一个限定的类型参数，请列出类型参数的名称，然后列出`extends`关键字，然后列出其上限（在本示例中为`Number`）。请注意，在这种情况下，`extends`通常用于表示“扩展”（如在类中）或“实现”（如在接口中）。

```
public class Box<T> {

    private T t;

    public void set(T t) {
        this.t = t;
    }

    public T get() {
        return t;
    }

    public <U extends Number> void inspect(U u){
        System.out.println("T: " + t.getClass().getName());
        System.out.println("U: " + u.getClass().getName());
    }

    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();
        integerBox.set(new Integer(10));
        integerBox.inspect("some text"); // error: this is still String!
    }
}
```

通过修改通用方法以包含此限定类型参数，由于我们的`inspect`调用仍包含`String`，因此编译现在将失败：

```
Box.java:21: <U>inspect(U) in Box<java.lang.Integer> cannot
  be applied to (java.lang.String)
                        integerBox.inspect("10");
                                   ^
1 error
```

除了限制可用于实例化泛型类型的类型之外，限定类型参数还允许你调用在范围中定义的方法：

```
public class NaturalNumber<T extends Integer> {

    private T n;

    public NaturalNumber(T n)  { this.n = n; }

    public boolean isEven() {
        return n.intValue() % 2 == 0;
    }

    // ...
}
```

`isEven`方法通过n调用`Integer`类中定义的`intValue`方法。

Multiple Bounds（多重限定）

前面的示例说明了使用带单个限定的类型参数，但是一个类型参数可以具有多个限定：

```
<T extends B1 & B2 & B3>
```

具有多个限定的类型变量是范围中列出的所有类型的子类型。如果范围之一是类，则必须首先指定它。例如：

```
Class A { /* ... */ }
interface B { /* ... */ }
interface C { /* ... */ }

class D <T extends A & B & C> { /* ... */ }
```

如果未首先指定绑定A，则会出现编译时错误：

```
class D <T extends B & A & C> { /* ... */ }  // compile-time error
```

Generic Methods and Bounded Type Parameters（通用方法和限定类型参数）

限定类型参数是实现通用算法的关键。考虑以下方法，该方法计算数组`T[]`中大于指定元素elem的元素数。

```
public static <T> int countGreaterThan(T[] anArray, T elem) {
    int count = 0;
    for (T e : anArray)
        if (e > elem)  // compiler error
            ++count;
    return count;
}
```

该方法的实现很简单，但是不能编译，因为大于运算符（`>`）仅适用于基本类型，例如`short`、`int`、`double`、`long`、`float`、`byte`和`char`。你不能使用`>`运算符比较对象。要解决此问题，请使用`Comparable`接口限定的类型参数：

```
public interface Comparable<T> {
    public int compareTo(T o);
}
```

结果代码将是：

```
public static <T extends Comparable<T>> int countGreaterThan(T[] anArray, T
elem) {
    int count = 0;
    for (T e : anArray)
        if (e.compareTo(elem) > 0)
            ++count;
    return count;
}
```

Generics, Inheritance, and Subtypes（泛型，继承和子类型）

众所周知，只要类型兼容，就可以将一种类型的对象分配给另一种类型的对象。例如，你可以将一个`Integer`分配给一个`Object`，因为`Object`是`Integer`的超类型之一：

```
Object someObject = new Object();
Integer someInteger = new Integer(10);
someObject = someInteger;   // OK
```

在面向对象的术语中，这称为“`is a`”关系。由于`Integer`是一种`Object`，因此允许分配。但是`Integer`也是`Number`的一种，因此以下代码也有效：

```
public void someMethod(Number n) { /* ... */ }

someMethod(new Integer(10));   // OK
someMethod(new Double(10.1));   // OK
```

泛型也是如此。你可以执行通用类型调用，将`Number`作为其类型参数传递，并且如果该参数与`Number`兼容，则可以随后进行`add`的任何后续调用：

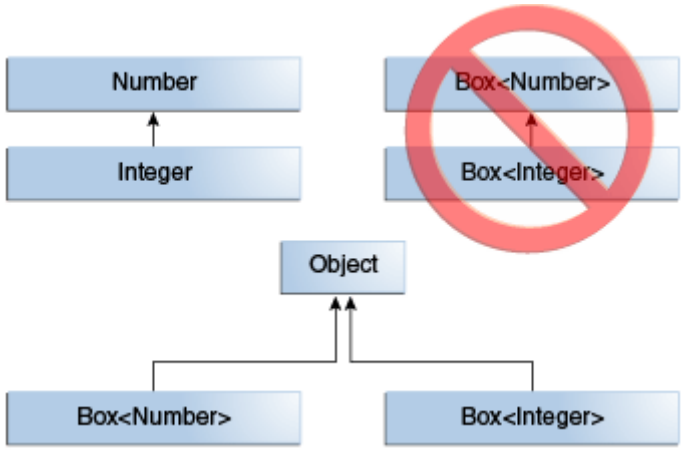
```
Box<Number> box = new Box<Number>();
box.add(new Integer(10));   // OK
box.add(new Double(10.1));  // OK
```

现在考虑以下方法：

```
public void boxTest(Box<Number> n) { /* ... */ }
```

它接受哪种类型？通过查看其签名，你可以看到它接受一个类型为`Box`的单个参数。但是，这是什么意思？如你所料，你是否允许传递`Box`或`Box`？答案是“否”，因为`Box`和`Box`不是`Box`的子类型。

在使用泛型进行编程时，这是一个常见的误解，但它是一个重要的概念。

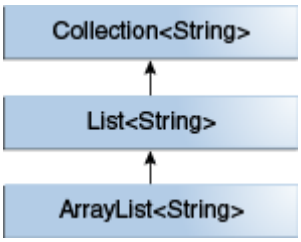


> 注意：给定两种具体的类型A和B（例如`Number`和`Integer`），无论A和B是否相关，`MyClass`与`MyClass`没有关系。`MyClass`和`MyClass`的公共父对象是`Object`。有关在类型参数相关时如何在两个泛型类之间创建类似子类型的关系的信息，请参见通配符和子类型。

Generic Classes and Subtyping（通用类和子类型）

你可以通过扩展或实现来泛型通用类或接口。一个类或接口的类型参数与另一类或接口的类型参数之间的关系由`extends`和`implements`子句确定。

以`Collections`类为例，`ArrayList`实现`List`，而`List`扩展`Collection`。因此，`ArrayList`是`List`的子类型，而`List`是`Collection`的子类型。只要你不改变类型参数，子类型关系就保留在类型之间。

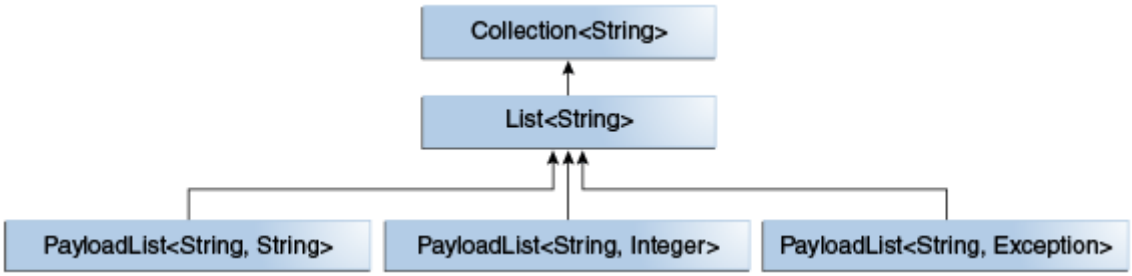


现在假设我们要定义自己的列表接口PayloadList，该接口将泛型P的可选值与每个元素相关联。它的声明可能看起来像：

```
interface PayloadList<E,P> extends List<E> {
  void setPayload(int index, P val);
}
```

PayloadList的以下参数化是`List`的子类型：

- `PayloadList<String,String>`
- `PayloadList<String,Integer>`
- `PayloadList<String,Exception>`



Type Inference（类型推断）

类型推断是Java编译器查看每个方法调用和相应声明以确定使调用适用的类型参数的能力。推断算法确定参数的类型，以及确定结果是否被分配或返回的类型（如果有）。最后，推断算法尝试找到与所有参数一起使用的最具体的类型。

为了说明最后一点，在下面的示例中，推断确定传递给`pick`方法的第二个参数的类型为`Serializable`：

```
static <T> T pick(T a1, T a2) { return a2; }
Serializable s = pick("d", new ArrayList<String>());
```

Type Inference and Generic Methods（类型推断和通用方法）

通用方法为你引入了类型推断，使你可以像调用普通方法一样调用通用方法，而无需在尖括号之间指定类型。考虑下面的示例BoxDemo，它需要Box类：

```
public class BoxDemo {

  public static <U> void addBox(U u,
      java.util.List<Box<U>> boxes) {
    Box<U> box = new Box<>();
    box.set(u);
    boxes.add(box);
  }

  public static <U> void outputBoxes(java.util.List<Box<U>> boxes) {
    int counter = 0;
    for (Box<U> box: boxes) {
      U boxContents = box.get();
      System.out.println("Box #" + counter + " contains [" +
             boxContents.toString() + "]");
      counter++;
    }
  }

  public static void main(String[] args) {
    java.util.ArrayList<Box<Integer>> listOfIntegerBoxes =
      new java.util.ArrayList<>();
    BoxDemo.<Integer>addBox(Integer.valueOf(10), listOfIntegerBoxes);
    BoxDemo.addBox(Integer.valueOf(20), listOfIntegerBoxes);
    BoxDemo.addBox(Integer.valueOf(30), listOfIntegerBoxes);
    BoxDemo.outputBoxes(listOfIntegerBoxes);
  }
}
```

以下是此示例的输出：

```
Box #0 contains [10]
Box #1 contains [20]
Box #2 contains [30]
```

通用方法`addBox`定义了一个名为U的类型参数。通常，Java编译器可以推断出通用方法调用的类型参数。因此，在大多数情况下，你不需要指定这些类型参数。例如，为了调用泛型方法`addBox`，你可以用一个类型见证来指定类型参数，如下所示。

```
BoxDemo.<Integer>addBox(Integer.valueOf(10), listOfIntegerBoxes);
```

或者，如果你省略了类型见证，Java编译器会自动推断出（从方法的参数（实参）中）类型参数是`Integer`。

```
BoxDemo.addBox(Integer.valueOf(20), listOfIntegerBoxes);
```

Type Inference and Instantiation of Generic Classes（泛型类的类型推断和实例化）

你可以用一组空的类型参数（`<>`）替换调用通用类的构造函数所需的类型参数，只要编译器可以从上下文中推断类型参数即可。这对尖括号被非正式地称为菱形。

例如，考虑以下变量声明：

```
Map<String, List<String>> myMap = new HashMap<String, List<String>>();
```

你可以用一组空的类型参数（`<>`）代替构造函数的参数化类型：

```
Map<String, List<String>> myMap = new HashMap<>();
```

请注意，要在泛型类实例化过程中利用类型推断的优势，必须使用菱形。在以下示例中，编译器生成未经检查的转换警告，因为`HashMap()`构造函数引用的是`HashMap`原始类型，而不是`Map>`类型：

```
Map<String, List<String>> myMap = new HashMap(); // unchecked conversion warning
```

Type Inference and Generic Constructors of Generic and Non-Generic Classes（泛型和非泛型类的类型推断和泛型构造函数）

请注意，构造函数在通用和非通用类中都可以是通用的（换句话说，声明自己的形式类型参数）。考虑以下示例：

```
class MyClass<X> {
  <T> MyClass(T t) {
    // ...
  }
}
```

考虑类MyClass的以下实例化：

```
new MyClass<Integer>("")
```

该语句创建参数化类型`MyClass`的实例；该语句为泛型类`MyClass`的形式类型参数X明确指定类型`Integer`。请注意，此泛型类的构造函数包含一个形式类型参数T。编译器会为该泛型类的构造函数的形式类型参数T推断`String`类型（因为该构造函数的实际参数是`String`对象）。

Java SE 7之前的发行版中的编译器能够推断泛型构造函数的实际类型参数，类似于泛型方法。但是，如果使用菱形（`<>`），则Java SE 7和更高版本中的编译器可以推断要实例化的泛型类的实际类型参数。考虑以下示例：

```
MyClass<Integer> myObject = new MyClass<>("");
```

在此示例中，编译器为通用类`MyClass`的形式类型参数X推断类型`Integer`。它为该泛型类的构造函数的形式类型参数T推断类型`String`。

> 注意：值得注意的是，推断算法仅使用调用参数，目标类型，并且可能使用明显的预期返回类型。推断算法不使用程序后面的结果。

Target Types（目标类型）

Java编译器利用目标类型来推断通用方法调用的类型参数。表达式的目标类型是Java编译器期望的数据类型，具体取决于表达式出现的位置。考虑方法`Collections.emptyList`，该方法声明如下：

```
static <T> List<T> emptyList();
```

考虑以下赋值语句：

```
List<String> listOne = Collections.emptyList();
```

该语句需要`List`的实例；此数据类型是目标类型。因为方法`emptyList`返回类型为`List`的值，所以编译器推断类型参数T必须为值`String`。这在Java SE 7和8中都可以使用。或者，你可以使用类型见证并按如下所示指定T的值：

```
List<String> listOne = Collections.<String>emptyList();
```

但是，在这种情况下，这不是必需的。不过，在其它情况下这是必要的。请考虑以下方法：

```
void processStringList(List<String> stringList) {
    // process stringList
}
```

假设你要使用空列表调用方法`processStringList`。在Java SE 7中，以下语句不会编译：

```
processStringList(Collections.emptyList());
```

Java SE 7编译器生成类似于以下内容的错误消息：

```
List<Object> cannot be converted to List<String>
```

编译器需要类型参数T的值，因此它以值`Object`开头。因此，对`Collections.emptyList`的调用返回类型为`List`的值，该值与方法`processStringList`不兼容。因此，在Java SE 7中，必须指定type参数值的值，如下所示：

```
processStringList(Collections.<String>emptyList());
```

这在Java SE 8中已经没有必要了，目标类型的概念已经扩展到包括方法参数，例如方法`processStringList`的参数。在这种情况下，`processStringList`需要一个`List`类型的参数。方法`Collections.emptyList`返回一个`List`的值，因此，使用`List`的目标类型，编译器会推断出类型参数T的值为`String`。因此，在Java SE 8中，编译器会编译出以下语句：

```
processStringList(Collections.emptyList());
```

有关更多信息，请参见Lambda表达式中的目标类型。

Wildcards（通配符）

在通用代码中，称为通配符的问号（`?`）表示未知类型。通配符可以在多种情况下使用：作为参数，字段或局部变量的类型；有时作为返回类型（尽管更具体的做法是更好的编程习惯）。通配符从不用作泛型方法调用，泛型类实例创建或超类型的类型参数。

以下各节将更详细地讨论通配符，包括上界通配符，下界通配符和通配符捕获。

Upper Bounded Wildcards（上限通配符）

你可以使用上限通配符来放宽对变量的限制。例如，假设你要编写一种适用于`List`，`List`和`List`的方法；可以使用上限通配符来实现。

要声明上限通配符，请使用通配符（`?`），后跟`extends`关键字，然后是其上限。请注意，在这种情况下，`extends`通常用于表示"扩展"（如在类中）或"实现"（如在接口中）。

要编写在`Number`类型的列表和`Number`的子类型（如`Integer`、`Double`和`Float`）上工作的方法，你会指定`List`。术语`List`比`List`更有限制性，因为前者只匹配`Number`类型的列表，而后者匹配`Number`类型的列表或其任何子类。

考虑以下处理方法：

```
public static void process(List<? extends Foo> list) { /* ... */ }
```

上限通配符，`，其中Foo是任何类型，匹配Foo和Foo的任何子类型。process`方法可以以类型Foo的形式访问列表元素。

```
public static void process(List<? extends Foo> list) {
    for (Foo elem : list) {
        // ...
    }
}
```

在`foreach`子句中，elem变量对列表中的每个元素进行迭代。在Foo类中定义的任何方法现在都可以在elem上使用。

`sumOfList`方法返回一个列表中的数字之和：

```
public static double sumOfList(List<? extends Number> list) {
    double s = 0.0;
    for (Number n : list)
        s += n.doubleValue();
    return s;
}
```

下面的代码，使用一个`Integer`对象的列表，打印出sum = 6.0。

```
List<Integer> li = Arrays.asList(1, 2, 3);
System.out.println("sum = " + sumOfList(li));
```

一个`Double`值的列表可以使用同样的`sumOfList`方法。下面的代码打印出sum = 7.0。

```
List<Double> ld = Arrays.asList(1.2, 2.3, 3.5);
System.out.println("sum = " + sumOfList(ld));
```

Unbounded Wildcards（无限通配符）

无界通配符类型使用通配符（`?`）来指定，例如`List`。这就是所谓的未知类型的列表。有两种情况下，无界通配符是一种有用的方法。

- 如果你正在编写一个可以使用`Object`类中提供的功能实现的方法。
- 当代码使用通用类中不依赖于类型参数的方法时。例如，`List.size`或`List.clear`。事实上，`Class`之所以这么经常使用，是因为`Class`中的大部分方法都不依赖于T。

考虑以下方法，`printList`：

```
public static void printList(List<Object> list) {
    for (Object elem : list)
        System.out.println(elem + " ");
    System.out.println();
}
```

`printList`的目标是打印任何类型的列表，但未能实现该目标（它仅打印`Object`实例的列表）；它不能打印`List`、`List`、`List`等，因为它们不是`List`的子类型。要编写通用的`printList`方法，请使用`List`：

```
public static void printList(List<?> list) {
    for (Object elem: list)
        System.out.print(elem + " ");
    System.out.println();
}
```

因为对于任何具体类型A，`List`是`List`的子类型，所以可以使用`printList`打印任何类型的列表：

```
List<Integer> li = Arrays.asList(1, 2, 3);
List<String>  ls = Arrays.asList("one", "two", "three");
printList(li);
printList(ls);
```

> 注意：本课的示例中均使用`Arrays.asList`方法。此静态工厂方法将转换指定的数组并返回固定大小的列表。

重要的是要注意`List`和`List`不同。你可以将`Object`或`Object`的任何子类型插入`List`。但是你只能将`null`插入`List`。通配符使用准则部分提供了有关如何确定在给定情况下应使用哪种通配符（如果有）的更多信息。

Lower Bounded Wildcards（下界通配符）

“上限通配符”部分显示，上限通配符将未知类型限制为特定类型或该类型的子类型，并使用`extends`关键字表示。以类似的方式，下限通配符将未知类型限制为特定类型或该类型的超类型。

下限通配符使用通配符（`?`）表示，后跟`super`关键字，后跟下限, 如：``

> 注意：你可以为通配符指定一个上限，也可以指定一个下限，但不能同时指定两者。

假设你要编写一个将`Integer`对象放入列表的方法。为了最大程度地提高灵活性，你希望该方法可用于`List`，`List`和`List`（可以容纳`Integer`值的任何内容）。

要编写对`Integer`的列表和`Integer`的超类型（如`Integer`、`Number`和`Object`）的方法，你可以指定`List`。术语`List`比`List`的限制性更强，因为前者只匹配`Integer`类型的列表，而后者则匹配作为`Integer`的超类型的任何类型的列表。

以下代码将数字1到10添加到列表的末尾：

```
public static void addNumbers(List<? super Integer> list) {
    for (int i = 1; i <= 10; i++) {
        list.add(i);
    }
}
```

通配符使用准则部分提供了有关何时使用上限通配符以及何时使用下限通配符的指南。

Wildcards and Subtyping（通配符和子类型）

如泛型，继承和子类型中所述，泛型类或接口不仅仅因为它们的类型之间存在关系而相关。但是，你可以使用通配符在通用类或接口之间创建关系。

给定以下两个常规（非泛型）类：

```
class A { /* ... */ }
```

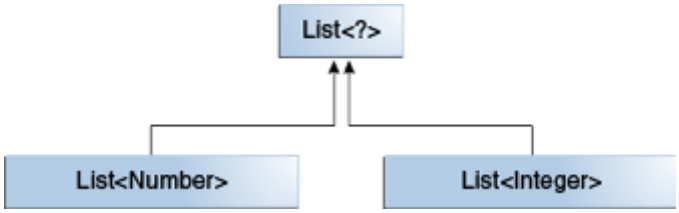
编写以下代码是合理的：

```
B b = new B();
A a = b;
复制代码
```

此示例显示常规类的继承遵循此子类型规则：如果B扩展了A，则类B是类A的子类型。此规则不适用于通用类型：

```
List<B> lb = new ArrayList<>();
List<A> la = lb;   // compile-time error
```

假定`Integer`是`Number`的子类型，则`List`和`List`之间是什么关系？

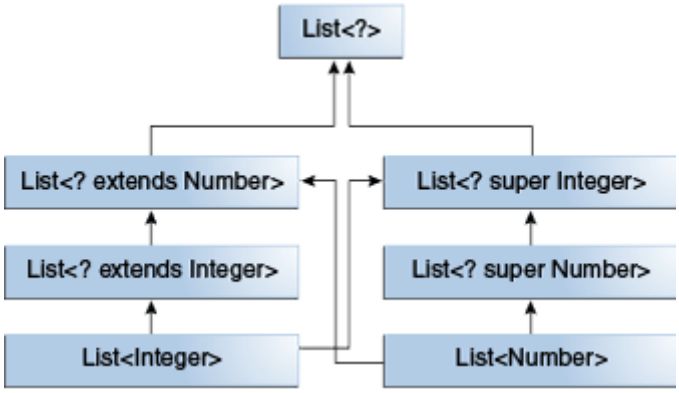


尽管`Integer`是`Number`的子类型，但`List`不是`List`的子类型，实际上，这两种类型无关。`List`和`List`的公共父级是`List`。

为了在这些类之间创建关系，以便代码可以通过`List`的元素访问`Number`的方法，请使用上限通配符：

```
List<? extends Integer> intList = new ArrayList<>();
List<? extends Number>  numList = intList;  // OK. List<? extends Integer>
is a subtype of List<? extends Number>
```

由于`Integer`是`Number`的子类型，并且numList是`Number`对象的列表，因此intList（一个`Integer`对象的列表）和numList之间现在存在关系。下图显示了使用上下限通配符声明的几个`List`类之间的关系。



通配符使用准则部分提供了有关使用上下限通配符的后果的更多信息。

Wildcard Capture and Helper Methods（通配符捕获和帮助方法）

在某些情况下，编译器会推断通配符的类型。例如，可以将列表定义为`List`，但是在评估表达式时，编译器会从代码中推断出特定类型。这种情况称为通配符捕获。

在大多数情况下，你无需担心通配符捕获，除非你看到包含短语“capture of”的错误消息。

WildcardError示例在编译时产生捕获错误：

```
import java.util.List;

public class WildcardError {

    void foo(List<?> i) {
        i.set(0, i.get(0));
    }
}
```

在此示例中，编译器将i输入参数处理为`Object`类型。当`foo`方法调用`List.set(int, E)`时，编译器无法确认要插入列表中的对象的类型，并产生错误。当发生这种类型的错误时，通常意味着编译器认为你正在将错误的类型分配给变量。为此，将泛型添加到Java语言中（以便在编译时强制类型安全）。

由Oracle的JDK 7 javac实现编译时，WildcardError示例将生成以下错误：

```
WildcardError.java:6: error: method set in interface List<E> cannot be
applied to given types;
    i.set(0, i.get(0));
     ^
  required: int,CAP#1
  found: int,Object
  reason: actual argument Object cannot be converted to CAP#1 by method
invocation conversion
  where E is a type-variable:
    E extends Object declared in interface List
  where CAP#1 is a fresh type-variable:
    CAP#1 extends Object from capture of ?
1 error
```

在此示例中，代码正在尝试执行安全操作，那么如何解决编译器错误？你可以通过编写捕获通配符的私有帮助器方法来修复它。在这种情况下，你可以通过创建私有帮助器方法`fooHelper`来解决此问题，如WildcardFixed中所示：

```
public class WildcardFixed {

    void foo(List<?> i) {
        fooHelper(i);
    }


    // Helper method created so that the wildcard can be captured
    // through type inference.
    private <T> void fooHelper(List<T> l) {
        l.set(0, l.get(0));
    }

}
```

由于使用了辅助方法，编译器在调用中使用推断来确定T是CAP＃1（捕获变量）。该示例现在可以成功编译。

按照约定，辅助方法通常命名为`originalMethodNameHelper`。

现在考虑一个更复杂的示例WildcardErrorBad：

```
import java.util.List;

public class WildcardErrorBad {

    void swapFirst(List<? extends Number> l1, List<? extends Number> l2) {
      Number temp = l1.get(0);
      l1.set(0, l2.get(0)); // expected a CAP#1 extends Number,
                            // got a CAP#2 extends Number;
                            // same bound, but different types
      l2.set(0, temp);      // expected a CAP#1 extends Number,
                            // got a Number
    }
}
```

在此的示例代码正在尝试不安全的操作。例如，考虑对swapFirst方法的以下调用：

```
List<Integer> li = Arrays.asList(1, 2, 3);
List<Double>  ld = Arrays.asList(10.10, 20.20, 30.30);
swapFirst(li, ld);
```

虽然`List`和`List`都满足了`List`的条件，但从`Integer`值列表中提取一个项并试图将其放入`Double`值列表中显然是不正确的。

使用Oracle的JDK javac编译器编译代码会产生以下错误：

```
WildcardErrorBad.java:7: error: method set in interface List<E> cannot be
applied to given types;
      l1.set(0, l2.get(0)); // expected a CAP#1 extends Number,
        ^
```

```
  required: int,CAP#1
  found: int,Number
  reason: actual argument Number cannot be converted to CAP#1 by method
invocation conversion
  where E is a type-variable:
    E extends Object declared in interface List
  where CAP#1 is a fresh type-variable:
    CAP#1 extends Number from capture of ? extends Number
WildcardErrorBad.java:10: error: method set in interface List<E> cannot be
applied to given types;
      l2.set(0, temp);      // expected a CAP#1 extends Number,
        ^
  required: int,CAP#1
  found: int,Number
  reason: actual argument Number cannot be converted to CAP#1 by method
invocation conversion
  where E is a type-variable:
    E extends Object declared in interface List
  where CAP#1 is a fresh type-variable:
    CAP#1 extends Number from capture of ? extends Number
WildcardErrorBad.java:15: error: method set in interface List<E> cannot be
applied to given types;
        i.set(0, i.get(0));
         ^
  required: int,CAP#1
  found: int,Object
  reason: actual argument Object cannot be converted to CAP#1 by method
invocation conversion
  where E is a type-variable:
    E extends Object declared in interface List
  where CAP#1 is a fresh type-variable:
    CAP#1 extends Object from capture of ?
3 errors
```

没有解决此问题的辅助方法，因为代码根本上是错误的。

Guidelines for Wildcard Use（通配符使用准则）

在学习使用泛型编程时，更令人困惑的方面之一是确定何时使用上限的通配符以及何时使用下限的通配符。此页面提供了一些在设计代码时要遵循的准则。

为了便于讨论，将变量视为提供以下两个功能之一将很有帮助：

“输入”变量：输入变量将数据提供给代码。想象一个具有两个参数的复制方法：`copy(src, dest)`。src参数提供要复制的数据，因此它是输入参数。

“输出”变量：输出变量保存要在其它地方使用的数据。在复制示例`copy(src, dest)`中，dest参数接受数据，因此它是输出参数。

当然，某些变量既用于“输入”又用于“输出”目的（准则中也解决了这种情况）。

在决定是否使用通配符以及哪种类型的通配符时，可以使用“输入”和“输出”原理。以下列表提供了要遵循的准则：

> 通配符准则：
>
> - 使用上限通配符定义输入变量，使用`extends`关键字。
> - 使用下限通配符定义输出变量，使用`super`关键字。

> - 如果可以使用`Object`类中定义的方法访问输入变量，请使用无界通配符（`?`）。
> - 如果代码需要同时使用输入和输出变量来访问变量，则不要使用通配符。

这些准则不适用于方法的返回类型。应该避免使用通配符作为返回类型，因为这会迫使程序员使用代码来处理通配符。

由`List`定义的列表可以被非正式地认为是只读的，但这并不是一个严格的保证。假设你有以下两个类。

```
class NaturalNumber {

    private int i;

    public NaturalNumber(int i) { this.i = i; }
    // ...
}

class EvenNumber extends NaturalNumber {

    public EvenNumber(int i) { super(i); }
    // ...
}
```

考虑以下代码：

```
List<EvenNumber> le = new ArrayList<>();
List<? extends NaturalNumber> ln = le;
ln.add(new NaturalNumber(35));  // compile-time error
```

因为`List`是`List`的一个子类型，所以可以将le赋给ln。但不能用ln将自然数添加到偶数列表中。可以对该列表进行以下操作。

- 可以添加`null`。
- 可以调用`clear`。
- 可以获取迭代器（`iterator`）和调用`remove`。
- 可以捕获通配符和写入从列表中读取的元素。

你可以看到由`List`定义的列表不是最严格意义上的只读，但你可能会这样想，因为你不能在列表中存储一个新的元素或改变一个现有的元素。

Type Erasure（类型擦除）

Java语言引入了泛型，以在编译时提供更严格的类型检查并支持泛型编程。 为了实现泛型，Java编译器将类型擦除应用于：

- 如果类型参数不受限制，则将通用类型中的所有类型参数替换为其边界（上下限）或`Object`。因此，产生的字节码仅包含普通的类，接口和方法。
- 必要时插入类型转换，以保持类型安全。
- 生成桥接方法以在扩展的泛型类型中保留多态。

类型擦除可确保不会为参数化类型创建新的类；因此，泛型不会产生运行时开销。

Erasure of Generic Types（泛型类型的擦除）

在类型擦除过程中，Java编译器将擦除所有类型参数，如果类型参数是有界的，则将每个参数替换为其第一个边界；如果类型参数是无界的，则将其替换为`Object`。

考虑以下表示单个链接列表中的节点的通用类：

```
public class Node<T> {

    private T data;
    private Node<T> next;

    public Node(T data, Node<T> next) {
        this.data = data;
        this.next = next;
    }

    public T getData() { return data; }
    // ...
}
```

由于类型参数T是无界的，因此Java编译器将其替换为`Object`：

```
public class Node {

    private Object data;
    private Node next;

    public Node(Object data, Node next) {
        this.data = data;
        this.next = next;
    }

    public Object getData() { return data; }
    // ...
}
```

在下面的示例中，通用Node类使用限定类型参数：

```
public class Node<T extends Comparable<T>> {

    private T data;
    private Node<T> next;

    public Node(T data, Node<T> next) {
        this.data = data;
        this.next = next;
    }

    public T getData() { return data; }
    // ...
}
```

Java编译器将绑定类型参数T替换为第一个绑定类`Comparable`：

```
public class Node {

    private Comparable data;
    private Node next;

    public Node(Comparable data, Node next) {
        this.data = data;
        this.next = next;
    }

    public Comparable getData() { return data; }
    // ...
}
```

Erasure of Generic Methods（通用方法的擦除）

Java编译器还会擦除通用方法参数中的类型参数。考虑以下通用方法：

```
// Counts the number of occurrences of elem in anArray.
//
public static <T> int count(T[] anArray, T elem) {
    int cnt = 0;
    for (T e : anArray)
        if (e.equals(elem))
            ++cnt;
        return cnt;
}
```

由于T是无界的，因此Java编译器将其替换为`Object`：

```
public static int count(Object[] anArray, Object elem) {
    int cnt = 0;
    for (Object e : anArray)
        if (e.equals(elem))
            ++cnt;
        return cnt;
}
```

假设定义了以下类：

```
class Shape { /* ... */ }
class Circle extends Shape { /* ... */ }
class Rectangle extends Shape { /* ... */ }
```

你可以编写一个通用方法来绘制不同的形状：

```
public static <T extends Shape> void draw(T shape) { /* ... */ }
```

Java编译器用Shape替换T：

```
public static void draw(Shape shape) { /* ... */ }
```

Effects of Type Erasure and Bridge Methods（类型擦除和桥接方法的影响）

有时类型擦除会导致可能无法预料的情况。以下示例显示了这种情况的发生方式。该示例（在“桥接方法”中进行了介绍）展示了编译器有时如何创建一个综合方法，称为桥接方法，作为类型擦除过程的一部分。

给定以下两类：

```
public class Node<T> {

    public T data;

    public Node(T data) { this.data = data; }

    public void setData(T data) {
        System.out.println("Node.setData");
        this.data = data;
    }
}

public class MyNode extends Node<Integer> {
    public MyNode(Integer data) { super(data); }

    public void setData(Integer data) {
        System.out.println("MyNode.setData");
        super.setData(data);
    }
}
```

考虑以下代码：

```
MyNode mn = new MyNode(5);
Node n = mn;            // A raw type - compiler throws an unchecked warning
n.setData("Hello");
Integer x = mn.data;    // Causes a ClassCastException to be thrown.
```

类型擦除后，此代码变为：

```
MyNode mn = new MyNode(5);
Node n = (MyNode)mn;         // A raw type - compiler throws an unchecked
warning
n.setData("Hello");
Integer x = (String)mn.data; // Causes a ClassCastException to be thrown.
```

执行代码时会发生以下情况：

- `n.setData("Hello");`导致在MyNode类的对象上执行`setData(Object)`方法（MyNode类继承了Node的`setData(Object)`）。
- 在`setData(Object)`的主体中，将n引用的对象的数据字段分配给`String`。
- 可以访问通过mn引用的同一对象的数据字段，并且该字段应该是整数（因为mn是MyNode，它是`Node`）。
- 尝试将`String`分配给`Integer`会导致Java编译器在分配时插入的强制转换导致`ClassCastException`。

Bridge Methods（桥接方法）

在编译扩展参数化类或实现参数化接口的类或接口时，作为类型擦除过程的一部分，编译器可能需要创建一个称为桥接方法的综合方法。你通常不必担心桥接方法，但是如果其中一个出现在堆栈跟踪中，你可能会感到困惑。

类型擦除后，Node和MyNode类变为：

```
public class Node {

    public Object data;

    public Node(Object data) { this.data = data; }

    public void setData(Object data) {
        System.out.println("Node.setData");
        this.data = data;
    }
}

public class MyNode extends Node {

    public MyNode(Integer data) { super(data); }

    public void setData(Integer data) {
        System.out.println("MyNode.setData");
        super.setData(data);
    }
}
```

类型擦除后，方法签名不匹配。Node方法变为`setData(Object)`，而MyNode方法变为`setData(Integer)`。因此，MyNode的`setData`方法不会覆盖Node的`setData`方法。

为了解决此问题并在类型擦除后保留泛型类型的多态性，Java编译器生成了一个桥接方法来确保子类型能够按预期工作。对于MyNode类，编译器为`setData`生成以下桥接方法：

```
class MyNode extends Node {

    // Bridge method generated by the compiler
    //
    public void setData(Object data) {
        setData((Integer) data);
    }

    public void setData(Integer data) {
        System.out.println("MyNode.setData");
        super.setData(data);
    }

    // ...
}
```

可以看到，在类型擦除后，MyNode类的桥接方法（`setData(Object)`）与Node类的`setData(Object)`方法具有相同的方法签名，它委托给原来的`setData(Integer)`方法。

Non-Reifiable Types（不可具体化类型）

“类型擦除”部分讨论了编译器删除与类型参数和类型参数有关的信息的过程。类型擦除的结果与变量参数（也称为varargs）方法有关，这些方法的varargs形式参数具有不可更改的类型。有关varargs方法的更多信息，请参见将信息传递给方法或构造方法中的任意参数数目。

此页面涵盖以下主题：

- 不可具体化类型。
- 堆污染。
- 具有不可具体化形式参数的Varargs方法的潜在漏洞。
- 防止使用不可具体化形式参数的Varargs方法发出警告。

Non-Reifiable Types（不可具体化类型）

具体化类型是其类型信息在运行时完全可用的类型。这包括基本类型，非通用类型，原始（raw）类型以及未绑定通配符的调用。

非具体化类型是指在编译时通过类型擦除法删除了信息的类型（对通用类型的调用没有被定义为非绑定通配符）。非具体化类型在运行时并不具备所有的信息。非具体化类型的例子是`List`和`List`；JVM在运行时无法区分这些类型。正如对通用类型的限制中所示，在某些情况下，非具体化类型不能使用：例如，在`instanceof`表达式中，或者作为数组中的元素。

Heap Pollution（堆污染）

当参数化类型的变量引用的对象不是该参数化类型的对象时，就会发生堆污染。如果程序执行某些操作会在编译时产生未经检查的警告，则会发生这种情况。如果在编译时（在编译时类型检查规则的范围内）或在运行时，无法确定涉及参数化类型的操作（例如，强制转换或方法调用）的正确性，则会生成未经检查的警告。例如，当混合原始（raw）类型和参数化类型时，或者执行未经检查的强制转换时，就会发生堆污染。

在正常情况下，当同时编译所有代码时，编译器会发出未经检查的警告，以引起你对潜在堆污染的注意。如果分别编译代码部分，则很难检测到堆污染的潜在风险。如果确保代码在没有警告的情况下进行编译，则不会发生堆污染。

Potential Vulnerabilities of Varargs Methods with Non-Reifiable Formal Parameters（具有不可具体化形式参数的Varargs方法的潜在漏洞）

包含vararg输入参数的泛型方法可能导致堆污染。

考虑以下ArrayBuilder类：

```
1  public class ArrayBuilder {
2
3    public static <T> void addToList (List<T> listArg, T... elements) {
4      for (T x : elements) {
5        listArg.add(x);
6      }
7    }
8
9    public static void faultyMethod(List<String>... l) {
10     Object[] objectArray = l;     // Valid
11     objectArray[0] = Arrays.asList(42);
12     String s = l[0].get(0);       // ClassCastException thrown here
13   }
14
15 }
```

以下示例HeapPollutionExample使用ArrayBuiler类：

```
public class HeapPollutionExample {

  public static void main(String[] args) {

    List<String> stringListA = new ArrayList<String>();
    List<String> stringListB = new ArrayList<String>();

    ArrayBuilder.addToList(stringListA, "Seven", "Eight", "Nine");
    ArrayBuilder.addToList(stringListB, "Ten", "Eleven", "Twelve");
    List<List<String>> listOfStringLists =
      new ArrayList<List<String>>();
    ArrayBuilder.addToList(listOfStringLists,
      stringListA, stringListB);

    ArrayBuilder.faultyMethod(Arrays.asList("Hello!"),
Arrays.asList("World!"));
  }
}
```

编译后，`ArrayBuilder.addToList`方法的定义会产生以下警告：

```
warning: [varargs] Possible heap pollution from parameterized vararg type T
```

当编译器遇到varargs方法时，它将varargs形式参数转换为数组。但是，Java编程语言不允许创建参数化类型的数组。在方法`ArrayBuilder.addToList`中，编译器将varargs形式参数`T ...`元素转换为形式参数`T[]`元素，即数组。但是，由于类型擦除，编译器将varargs形式参数转换为`Object[]`元素。因此，存在堆污染的可能性。

以下语句将varargs形式参数l分配给对象数组objectArgs：

```
Object[] objectArray = l;
```

该语句可能会导致堆污染。可以将与varargs形式参数l的参数化类型匹配的值分配给变量objectArray，从而可以将其分配给l。但是，编译器不会在此语句上生成未经检查的警告。当编译器将varargs形式参数`List... l`转换为形式参数`List[] l`时，已经生成了警告。此声明有效；变量l具有类型`List[]`，它是`Object[]`的子类型。

因此，如果将任何类型的`List`对象分配给objectArray数组的任何数组组件，则编译器不会发出警告或错误，如以下语句所示：

```
objectArray[0] = Arrays.asList(42);
```

该语句将`List`对象分配给objectArray数组的第一个数组组件，该`List`对象包含一个`Integer`类型的对象。

假设你使用以下语句调用`ArrayBuilder.faultyMethod`：

```
ArrayBuilder.faultyMethod(Arrays.asList("Hello!"), Arrays.asList("World!"));
```

在运行时，JVM在以下语句中引发`ClassCastException`：

```
// ClassCastException thrown here
String s = l[0].get(0);
```

存储在变量l的第一个数组组件中的对象的类型为`List`，但是此语句期望使用类型为`List`的对象。

Prevent Warnings from Varargs Methods with Non-Reifiable Formal Parameters（防止使用不可具体化形式参数的Varargs方法发出警告）

如果你声明具有参数化类型参数的varargs方法，并确保由于对varargs形式参数的处理不当，该方法的主体不会引发`ClassCastException`或其它类似的异常，则可以避免警告编译器通过为静态和非构造方法声明添加以下注解，为此类varargs方法生成：

```
@SafeVarargs
```

`@SafeVarargs`注解是该方法契约的书面部分；该注解断言该方法的实现不会不适当地处理varargs形式参数。

尽管不太理想，但也可以通过在方法声明中添加以下内容来抑制此类警告：

```
@SuppressWarnings({"unchecked", "varargs"})
```

但是，这种方法不能抑制从该方法的调用站点生成的警告。如果你不熟悉`@SuppressWarnings`语法，请参阅注解。

Restrictions on Generics（对泛型的限制）

为了有效地使用Java泛型，必须考虑以下限制：

- 无法实例化具有基本类型的泛型类型。
- 无法创建类型参数的实例。
- 无法声明类型为类型参数的静态字段。
- 无法将Casts或instanceof与参数化类型一起使用。
- 无法创建参数化类型的数组。
- 无法创建，捕获或抛出参数化类型的对象。
- 无法重载每个重载的形式参数类型都擦除为相同原始（raw）类型的方法。

Cannot Instantiate Generic Types with Primitive Types（无法实例化具有基本类型的泛型类型）

考虑以下参数化类型：

```
class Pair<K, V> {

    private K key;
    private V value;

    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    // ...
}
```

创建对对象时，不能用基本类型替换类型参数K或V：

```
Pair<int, char> p = new Pair<>(8, 'a');  // compile-time error
```

你只能将非基本类型替换为类型参数K和V：

```
Pair<Integer, Character> p = new Pair<>(8, 'a');
```

请注意，Java编译器自动将`8`装箱为`Integer.valueOf(8)`，将`'a'`装箱为`Character('a')`：

```
Pair<Integer, Character> p = new Pair<>(Integer.valueOf(8), new
Character('a'));
```

有关自动装箱的更多信息，请参阅数字和字符串课程中的自动装箱和拆箱。

Cannot Create Instances of Type Parameters（无法创建类型参数的实例）

你不能创建类型参数的实例。例如，以下代码会导致编译时错误：

```
public static <E> void append(List<E> list) {
    E elem = new E();  // compile-time error
    list.add(elem);
}
```

解决方法是，可以通过反射创建类型参数的对象：

```
public static <E> void append(List<E> list, Class<E> cls) throws Exception {
    E elem = cls.newInstance();   // OK
    list.add(elem);
}
```

你可以按以下方式调用`append`方法：

```
List<String> ls = new ArrayList<>();
append(ls, String.class);
```

Cannot Declare Static Fields Whose Types are Type Parameters（无法声明类型为类型参数的静态字段）

类的静态字段是该类的所有非静态对象共享的类级别变量。因此，不允许使用类型参数的静态字段。考虑以下类别：

```
public class MobileDevice<T> {
    private static T os;

    // ...
}
```

如果允许使用类型参数的静态字段，那么以下代码将被混淆：

```
MobileDevice<Smartphone> phone = new MobileDevice<>();
MobileDevice<Pager> pager = new MobileDevice<>();
MobileDevice<TabletPC> pc = new MobileDevice<>();
```

因为静态字段os由Smartphone、Pager和TabletPC共享，所以os的实际类型是什么？它不能同时是Smartphone、Pager和TabletPC。因此，你无法创建类型参数的静态字段。

Cannot Use Casts or instanceof With Parameterized Types（无法将Casts或instanceof与参数化类型一起使用）

因为Java编译器会擦除通用代码中的所有类型参数，所以你无法验证在运行时使用的是通用类型的参数化类型：

```
public static <E> void rtti(List<E> list) {
    if (list instanceof ArrayList<Integer>) {  // compile-time error
        // ...
    }
}
```

传递给`rtti`方法的参数化类型的集合是：

```
S = { ArrayList<Integer>, ArrayList<String> LinkedList<Character>, ... }
```

运行时不跟踪类型参数，因此无法区分`ArrayList`和`ArrayList`之间的区别。你最多可以做的是使用无界通配符来验证列表是否为ArrayList：

```
public static void rtti(List<?> list) {
    if (list instanceof ArrayList<?>) {  // OK; instanceof requires a reifiable type
        // ...
    }
}
```

通常，除非使用不受限制的通配符对其进行参数化，否则无法将其转换为参数化类型。例如：

```
List<Integer> li = new ArrayList<>();
```

但是，在某些情况下，编译器知道类型参数始终有效并允许强制转换。例如：

```
List<String> l1 = ...;
ArrayList<String> l2 = (ArrayList<String>)l1;  // OK
```

Cannot Create Arrays of Parameterized Types（无法创建参数化类型的数组）

你不能创建参数化类型的数组。例如，以下代码无法编译：

```
List<Integer>[] arrayOfLists = new List<Integer>[2];  // compile-time error
```

以下代码说明了将不同类型插入到数组中时发生的情况：

```
Object[] strings = new String[2];
strings[0] = "hi";   // OK
strings[1] = 100;    // An ArrayStoreException is thrown.
```

如果你对通用列表尝试相同的操作，则会出现问题：

```
Object[] stringLists = new List<String>[];  // compiler error, but pretend it's allowed
stringLists[0] = new ArrayList<String>();   // OK
stringLists[1] = new ArrayList<Integer>();  // An ArrayStoreException should be thrown,
                                            // but the runtime can't detect it.
```

如果允许参数化列表的数组，那么前面的代码将无法抛出所需的`ArrayStoreException`。

Cannot Create, Catch, or Throw Objects of Parameterized Types（无法创建，捕获或抛出参数化类型的对象）

泛型类不能直接或间接扩展`Throwable`类。例如，以下类将无法编译：

```
// Extends Throwable indirectly
class MathException<T> extends Exception { /* ... */ }    // compile-time error

// Extends Throwable directly
class QueueFullException<T> extends Throwable { /* ... */ // compile-time error
```

方法无法捕获类型参数的实例：

```
public static <T extends Exception, J> void execute(List<J> jobs) {
    try {
        for (J job : jobs)
            // ...
    } catch (T e) {   // compile-time error
        // ...
    }
}
```

但是，你可以在`throws`子句中使用类型参数：

```
class Parser<T extends Exception> {
    public void parse(File file) throws T {     // OK
        // ...
    }
}
```

Cannot Overload a Method Where the Formal Parameter Types of Each Overload Erase to the Same Raw Type（无法重载每个重载的形式参数类型都擦除为相同原始（raw）类型的方法）

一个类不能有两个重载的方法，这些方法在类型擦除后将具有相同的签名。

```
public class Example {
    public void print(Set<String> strSet) { }
    public void print(Set<Integer> intSet) { }
}
```

重载将共享相同的类文件表示形式，并且将生成编译时错误。